On Squaring and Multiplying Large Integers

Dan Zuras
Hewlett-Packard Co.

ABSTRACT

Methods of squaring large integers are discussed. The obvious $O(n^2)$ method turns out to be best for small numbers. The existing $O(n^{\log 3/\log 2}) \approx O(n^{1.585})$ method becomes better as the numbers get bigger. New methods that are $O(n^{\log 5/\log 3}) \approx O(n^{1.465})$ and $O(n^{\log 7/\log 4}) \approx O(n^{1.404})$ are presented. All of these methods can be generalized to multiply and turn out to be faster than an FFT multiply for numbers that can be quite large (>3,000,000 bits). Squaring seems to be fundamentally faster than multiply but it is shown that $T_{multiply} \leq 2T_{square} + O(n)$.

1. Introduction

In his classic work *The Art of Computer Programming*[4], Don Knuth asked the question "How Fast Can We Multiply?" His answer was far more comprehensive than this one will be. In fact, much of the work presented in this paper is only a minor modification of the work of A. L. Toom[6].

In this paper, some of the simpler methods of squaring are shown to be best in a surprisingly large number of cases. Some new methods, and their generalizations to multiply, will also be discussed which are useful out to quite large numbers. In light of these results it seems that methods such as the Schönhage and Strassen FFT multiply[5], while of theoretical interest, may never be best for any reasonably sized numbers.

2. The Problem

The problem to be discussed here is how to find the best (fastest) way to square large numbers in software. The methods presented here were implemented in C and assembly language on an HP-9000/720 but the general observations and conclusions should be true in wider areas of application.

The approach used was to write a collection of routines for squaring w-word numbers producing $2w$-word results for various specific values of w. These were general purpose routines in the sense that w was a parameter.

In the course of writing these routines, it became clear that some common operations on large integers stored as arrays of unsigned 32-bit words would be needed(1). These operations were written in assembly language and performed the functions detailed in Table 1.

Given operands a[0,w-1], b[0,w-1], producing result r[0,w-1]	
vzconst(r,w,c)	r[0,w-1] = ccc...c (n c's)
vzcopy(r,w,a)	r[0,w-1] = a[0,w-1]
vzneg(r,w,a)	r[0,w-1] = -a[0,w-1]
vzshlM(r,w,a,c)	r[0,w-1] = a[0,w-1]|c<<M for M in [1,2,4,8,16,31]
vzshr(r,w,a,M)	r[0,w-1] = a[0,w-1]>>M for M in [0,31]
vzadd(r,w,a,b)	r[0,w-1] = a[0,w-1] + b[0,w-1]
vzsub(r,w,a,b)	r[0,w-1] = a[0,w-1] - b[0,w-1]
Cout = vzaddwco(r,w,a,b)	r[0,w-1] + Cout = a[0,w-1] + b[0,w-1]
vzx5(r,w,a)	r[0,w-1] = 5*a[0,w-1]
vzx17(r,w,a)	r[0,w-1] = 17*a[0,w-1]
vzx257(r,w,a)	r[0,w-1] = 257*a[0,w-1]
vzx65537(r,w,a)	r[0,w-1] = 65537*a[0,w-1]
t = vzge(b,w,a)	t = (b[0,w-1] ≥ a [0,w-1])

Table 1:

A collection of routines for squaring "small" integers were also written in assembly. These provided a basis upon which the larger routines could be

(1) These arrays were oriented "big-endian". That is, the most significant word of a[0,w-1] was stored in a[0] and the least significant word was stored in a[w-1]. While the author preferred "little-endian", both C and the HP-9000 PA-RISC architecture disagreed. They won, in the end.

1063-6889/93 $03.00 © 1993 IEEE

built.

All other routines were written in C and compiler optimized.

Actual running time was measured for w in the range 1 to 100,000 words (3,000,000 bits) by counting cycles on a 50MHz HP-9000/720.

3. Finding the Best

We will define $T_i(w)$ as the time required for method i to square a number of length w.

Method i is considered *best* for some length w if

$T_i(w) \le T_j(w'), \forall j \forall w' \ge w.$

That is, a method is best for a given length if there is no faster way of squaring numbers at least as large as this one.

4. Method 1: The n^2 Basis

To start the ball rolling, basis routines for operating on small integers were written using the obvious method.

Roughly speaking, if the algorithm for a multiply is

```
FOR i = w-1 to 0:
    FOR j =w-1 to 0:
        (r[i+j,i+j+1] and CarryOut)
                              += a[i]×b[j],
```

then squaring can be made strictly faster than multiply by accumulating the off-diagonal elements separately and combining them with the diagonals according to the formula

$Result = 2Offdiagonals + Diagonals$

Obviously, the running time, $T_1(w)$, increases quadratically. $T_1(w)$, in cycles, is well approximated by the rationalized least squares formula

$T_1(w) = ((21w + 2371/17)w - 295)/5.$

These routines turned out to be best for all w up to 25.

5. The 2-Way Method

Karatsuba and Ofman[3] presented a method of squaring a number in less than n^2 time based on the formula[2]

$$(A_1x + A_0)^2 = A_1^2(x^2 - x) + (A_1 + A_0)^2x + A_0^2(1 - x)$$

where $x = 2^n$.

Knuth presented a minor improvement on this based on the slightly different formula

$$(A_1x + A_0)^2 = A_1^2(x^2 + x) - (A_1 - A_0)^2x + A_0^2(x + 1)$$

Both of these may be regarded as special cases of solving for the C_i in the equation

$$(A_1x + A_0)^2 = C_2x^2 + C_1x + C_0$$

Karatsuba and Ofman solve this equation by letting x take on the values $\{\infty, 1, 0\}$ [3]. Rather than express this system in the usual polynomial form, I will express it as a linear transformation of the operand in the following way

$$\begin{bmatrix} V_2 \\ V_1 \\ V_0 \end{bmatrix} = \begin{bmatrix} A_1 \\ A_1 + A_0 \\ A_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} A_1 \\ A_0 \end{bmatrix}$$

After squaring each element of the vector, the solution to the resulting system

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} S_2 \\ S_1 \\ S_0 \end{bmatrix} = \begin{bmatrix} A_1^2 \\ (A_1 + A_0)^2 \\ A_0^2 \end{bmatrix}$$

may be expressed as

$$\begin{bmatrix} C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_2 \\ S_1 \\ S_0 \end{bmatrix}$$

Knuth solves the equation by letting x take on the values $\{\infty, -1, 0\}$. A similar transformation

$$\begin{bmatrix} V_2 \\ V_1 \\ V_0 \end{bmatrix} = \begin{bmatrix} A_1 \\ A_1 - A_0 \\ A_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} A_1 \\ A_0 \end{bmatrix}$$

results, after squaring, in a similar system

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} S_2 \\ S_1 \\ S_0 \end{bmatrix} = \begin{bmatrix} A_1^2 \\ (A_1 - A_0)^2 \\ A_0^2 \end{bmatrix}$$

---

(2) I will *not*, however, inflict the "big-endian" notation on the reader. Here, I will use the somewhat more conventional notation of mathematics. That is, "little-endian". So there.

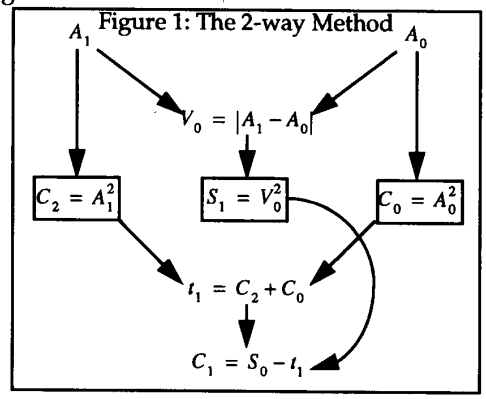
(3) In this context, $x = \infty$ corresponds to $\lim_{x \to \infty} (A_1x + A_0)^2/x^2 = \lim_{x \to \infty} (C_2x^2 + C_1x + C_0)/x^2$.

which has the solution

$$\begin{bmatrix} C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_2 \\ S_1 \\ S_0 \end{bmatrix}$$

This latter solution may be diagrammed as in Figure 1.

Figure 1: The 2-way Method



Assuming that the time for an add of length w is $O(w)$, the time to square a number of length $2w$ would be

$$T_2(2w) \approx 3T_{best}(w) + O(w)$$

The overhead involved, one add before the squaring step and two after, is strictly larger than the overhead for an n^2 method. Thus, although an n^2 squaring of length $2w$ would take

$$T_1(2w) \approx 4T_{best}(w) + O(w),$$

there is a definite crossover point as w increases where $T_1(w)$ is roughly the same as $T_2(w)$. Below this point the n^2 method wins because of its simplicity and above this point the 2-way method wins because of its better asymptotic behavior.

Figure 2 illustrates this by plotting the ratio $T_2(w)/T_1(w)$ for some 2-way routines.

In the limit, the time of the 2-way method triples each time the size of the problem doubles. Therefore, its time increases as $w^{\log 3/\log 2} \approx w^{1.585}$.

The running time of the 2-way routines (in cycles) is approximated by the rationalized formula

$$T_2(w) \approx (223w^{\log 3/\log 2} - 456w + 3341)/8.$$

This method can be turned into a multiply algorithm by replacing the S_i with the P_i in the formula

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} P_2 \\ P_1 \\ P_0 \end{bmatrix} = \begin{bmatrix} A_1B_1 \\ (A_1 - A_0)(B_1 - B_0) \\ A_0B_0 \end{bmatrix}$$

and doing the implied extra work.

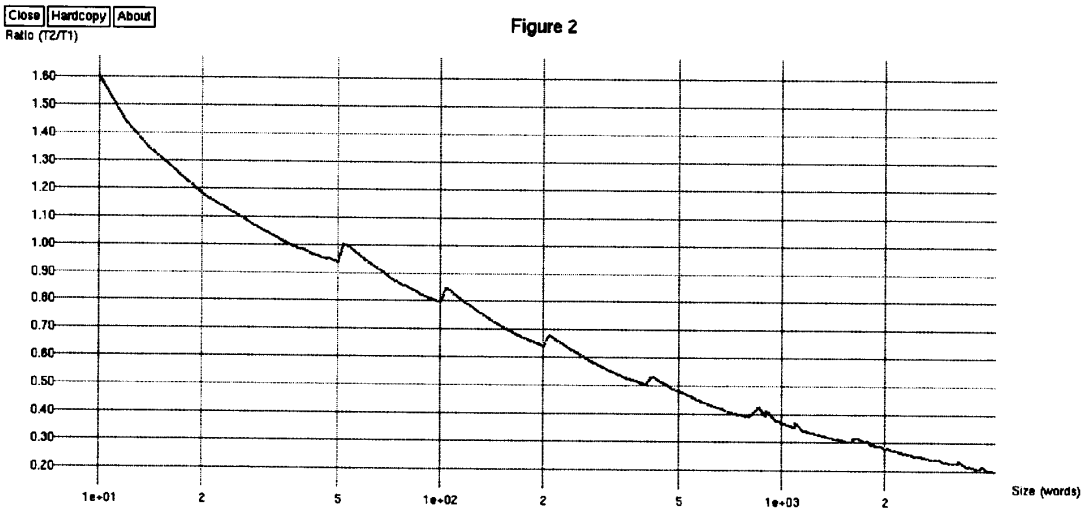
6. The General Idea

We now have enough background to show the general idea behind these methods.

In each case, the operand will be split into k parts A_j. Those parts will be transformed via a Q_{ij} (derived from the original polynomial) into m parts $V_i = Q_{ij}A_j$. Then, each of the V_i is squared with the best method available, producing $S_i = V_i^2$.

Now, the solution to the system $M_{ij}C_j = S_i$ (de-

Figure 2



rived from the square of the original polynomial) can be expressed as $C_i = \Sigma_{ik}^{-1} R_{kj} S_j$ with all the coefficients in the integers. (Up to now, Σ_{ik}^{-1}, which expresses the integer divides, has not been needed as it was the identity matrix.) Finally, the C_i are shifted and added up to produce the result.

7. The 3-Way Method

Toom[6] showed that circuits could be constructed for squaring integers where the size was bounded by $O(nc^{\sqrt{\log n}})$ and the delay was bounded by $O(c^{\sqrt{\log n}})$.

Cook[2] showed that an algorithm for squaring integers could be realized which had a running time of $O(n2^{5\sqrt{\log n}})$.

This algorithm, which is actually a collection of algorithms, has come to be known as the Toom-Cook method.

The first method in this collection is equivalent to Karatsuba's method.

The second method divides a number into three parts and involves solving for the C_i in the equation

$$(A_2x^2 + A_1x + A_0)^2 = C_4x^4 + C_3x^3 + C_2x^2 + C_1x + C_0$$

Cook solves this equation by letting x take on the values $\{4, 3, 2, 1, 0\}$ in the polynomial, leading to the transformation

$$\begin{bmatrix} V_4 \\ V_3 \\ V_2 \\ V_1 \\ V_0 \end{bmatrix} = \begin{bmatrix} 16 & 4 & 1 \\ 9 & 3 & 1 \\ 4 & 2 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} A_2 \\ A_1 \\ A_0 \end{bmatrix}$$

which, after squaring, yields the system

$$\begin{bmatrix} 256 & 64 & 16 & 4 & 1 \\ 81 & 27 & 9 & 3 & 1 \\ 16 & 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C_4 \\ C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix} = \begin{bmatrix} (16A_2 + 4A_1 + A_0)^2 \\ (9A_2 + 3A_1 + A_0)^2 \\ (4A_2 + 2A_1 + A_0)^2 \\ (A_2 + A_1 + A_0)^2 \\ A_0^2 \end{bmatrix}$$

the solution of which I will express as

$$\begin{bmatrix} C_4 \\ C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} 24 & 0 & 0 & 0 & 0 \\ 0 & 12 & 0 & 0 & 0 \\ 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 12 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & -4 & 6 & -4 & 1 \\ -3 & 14 & -24 & 18 & -5 \\ 11 & -56 & 114 & -104 & 35 \\ -3 & 16 & -36 & 48 & -25 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix}$$

The diagonal terms in the first matrix are the denominators in divides of intermediate results. In practice, there turn out to be four divides by 3 since the rest amounts to shifts of 2 or 3 bits.

Knuth solves this with x taken from $\{2, 1, 0, -1, -2\}$ leading to the transformation

$$\begin{bmatrix} V_4 \\ V_3 \\ V_2 \\ V_1 \\ V_0 \end{bmatrix} = \begin{bmatrix} 4 & 2 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & -1 & 1 \\ 4 & -2 & 1 \end{bmatrix} \begin{bmatrix} A_2 \\ A_1 \\ A_0 \end{bmatrix}$$

and the system

$$\begin{bmatrix} 16 & 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 16 & -8 & 4 & -2 & 1 \end{bmatrix} \begin{bmatrix} C_4 \\ C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix} = \begin{bmatrix} (4A_2 + 2A_1 + A_0)^2 \\ (A_2 + A_1 + A_0)^2 \\ A_0^2 \\ (A_2 - A_1 + A_0)^2 \\ (4A_2 - 2A_1 + A_0)^2 \end{bmatrix}$$

which has the somewhat more symmetrical solution

$$\begin{bmatrix} C_4 \\ C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} 24 & 0 & 0 & 0 & 0 \\ 0 & 12 & 0 & 0 & 0 \\ 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 12 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & -4 & 6 & -4 & 1 \\ 1 & -2 & 0 & 2 & -1 \\ -1 & 16 & -30 & 16 & -1 \\ -1 & 8 & 0 & -8 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix}$$

Although both of these solutions look similar, Knuth's is somewhat better than the previous one.

The matrix expression of this solution hides the information that one can construct this solution with relatively few primitive operations (add, subtract, shift, and the like).

If, however, one chooses x from the set $\{\infty, 2, 1, 1/2, 0\}$ [4], then

(4) In general, $x = p/q$ corresponds to $q^{2n}(A_n(p/q)^n + \ldots + A_0)^2 = q^{2n}(C_{2n}(p/q)^{2n} + \ldots + C_0)$.

$$\begin{bmatrix} V_4 \\ V_3 \\ V_2 \\ V_1 \\ V_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 2 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} A_2 \\ A_1 \\ A_0 \end{bmatrix}$$

and

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 16 & 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C_4 \\ C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix} = \begin{bmatrix} A_2^2 \\ (4A_2 + 2A_1 + A_0)^2 \\ (A_2 + A_1 + A_0)^2 \\ (A_2 + 2A_1 + 4A_0)^2 \\ A_0^2 \end{bmatrix}$$

have the solution

$$\begin{bmatrix} C_4 \\ C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -21 & 2 & -12 & 1 & -6 \\ 7 & -1 & 10 & -1 & 7 \\ -6 & 1 & -12 & 2 & -21 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix}$$

This solution has only 2 divides by 3 and 3 shifts as well as much less overhead in the computation of the intermediate results.

The squaring method implied by this solution is shown in Figure 3.

All of the operations shown can be made $O(w)$. The three shifts can be done separately or as part of the combined adds that create the final result. That leaves the divides by 3.

While a divide by 3 can be done in $O(w)$ time, the actual method chosen, in spite of being asymptotically $O(w \log w)$, was believed to be faster.

This method, which actually is a divide by -3, involves only adds and shifts. Assuming that the number in question is in fact a multiple of 3, we have

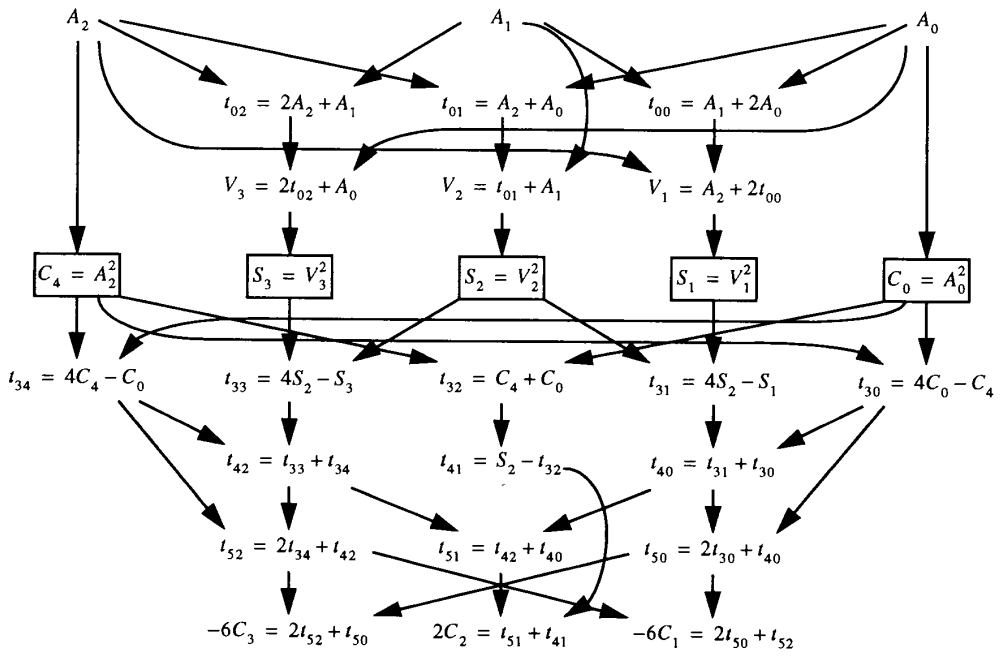
$$4(3k) + 3k \rightarrow 15k = 16k - k$$

$$16(15k) + 15k \rightarrow 255k = 256k - k$$

$$2^8((2^8 - 1)k) + (2^8 - 1)k \rightarrow (2^{16} - 1)k$$

$$2^{16}((2^{16} - 1)k) + (2^{16} - 1)k \rightarrow (2^{32} - 1)k$$

Figure 3: The 3-way Method



$2^{32}((2^{32}-1)k)+(2^{32}-1)k \to (2^{64}-1)k$

$2^{64}((2^{64}-1)k)+(2^{64}-1)k \to (2^{128}-1)k$

...

With sufficiently many steps one can, by discarding the $2^{large}k$ term, divide a number by -3.

Assuming that the time for operations of length w is $O(w)$, the time to square a number of length $3w$ would be

$$T_3(3w) \approx 5T_{best}(w) + O(w)$$

This suggests that the asymptotic running time of the 3-way method increases as $w^{\log 5/\log 3} \approx w^{1.465}$.

The improvement is not nearly so dramatic as the improvement in the 2-way method over the n^2 method. The advantages of more complex routines will be even less dramatic for much more work.

The running time of the 3-way method is approximated by the formula

$$T_3(w) \approx (1370w^{\log 5/\log 3} - 8718w + 457414)/17$$

As before, to turn this method into a multiply, one duplicates those operations before the squaring steps and multiplies the corresponding terms instead of squaring them.

8. Running Time of Toom-Cook Style Methods

The running time of a Toom-Cook, Knuth, & k-way methods will have three components: $m = 2k-1$ squarings of components of size w/k; some $O(w)$ overhead; and some fixed overhead.

Assuming that one starts with a basis routine of length w_0 that takes t_0 time, we have

$$T_k(w_0) = t_0$$

$$T_k(k^{r+1}w_0) = mT_k(k^r w_0) + c_1 k^r w_0 + c_0$$

$$T_k(kw_0) = mt_0 + c_1 w_0 + c_0$$

$$T_k(k^2 w_0) = m^2 t_0 + c_1 w_0 (m+k) + c_0 (m+1)$$

$$T_k(k^3 w_0) = m^3 t_0 + c_1 w_0 (m^2 + mk + k^2) + c_0 (m^2 + m + 1)$$

...

$$T_k(k^r w_0) = m^r t_0 + c_1 w_0 \frac{m^r - k^r}{m-k} + c_0 \frac{m^r - 1}{m-1}$$

Rearranging terms in this last equation gives

$$T_k(k^r w_0) = (t_0 + \frac{c_1 w_0}{m-k} + \frac{c_0}{m-1}) m^r - \frac{c_1 w_0 k^r}{m-k} - \frac{c_0}{m-1}$$

or, in terms of $w = k^r w_0$ becomes

$$T_k(w) = (t_0 + \frac{c_1 w_0}{m-k} + \frac{c_0}{m-1}) (\frac{w}{w_0})^{\frac{\log m}{\log k}} - \frac{c_1 w}{m-k} - \frac{c_0}{m-1}$$

(If a portion of the overhead takes $O(w \log w)$ time then this formula generalizes to

$$T_k(w) = A (\frac{w}{w_0})^{\frac{\log m}{\log k}} - \frac{c_2 kw \log w}{m-k} - C\frac{w}{m-k} - \frac{c_0}{m-1}$$

where

$$A = t_0 + (\frac{c_2 k \log k}{m-k} + c_2 \log w_0 + c_1) \frac{w_0}{m-k} + \frac{c_0}{m-1}$$

and

$$C = \frac{c_2 k \log k}{m-k} - c_2 (k-1) \log w_0 + c_1 .$$

But I have found that the simpler formula

$$T_k(w) = A' (\frac{w}{w_0})^{\frac{\log m}{\log k}} - B'w - C'$$

is more than adequate to represent the running time of these routines in the regions of interest.)

9. The 4-Way Method and Beyond

Extension to a k-way method involves solving for the C_i in the equation

$$(A_k x^k + \ldots + A_0)^2 = C_{2k} x^{2k} + \ldots + C_0$$

Cook would solve this equation by letting x take on the values $\{0, \ldots, 2k\}$.

Knuth would solve this equation by letting x take on the values $\{-k, \ldots, k\}$.

But if one chooses x from the set $\{1, \infty, 0, 2, \frac{1}{2}, -2, -\frac{1}{2}, 3, \frac{1}{3}, -3, -\frac{1}{3}, \frac{3}{2}, \frac{2}{3}, \ldots\}$, that is, the set of small rational numbers, $\pm\frac{p}{q}$, such that the $GCD(p, q) = 1$, the resulting system of equations is very symmetrical and the algorithm is simpler.

I will illustrate the various methods for $k = 4$.

Cook:

$$Q = \begin{bmatrix} 216 & 36 & 6 & 1 \\ 125 & 25 & 5 & 1 \\ 64 & 16 & 4 & 1 \\ 27 & 9 & 3 & 1 \\ 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} 46656 & 7776 & 1296 & 216 & 36 & 6 & 1 \\ 15625 & 3125 & 625 & 125 & 25 & 5 & 1 \\ 4096 & 1024 & 256 & 64 & 16 & 4 & 1 \\ 729 & 243 & 81 & 27 & 9 & 3 & 1 \\ 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 720 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 240 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 144 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 48 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 360 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 60 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R = \begin{bmatrix} 1 & -6 & 15 & -20 & 15 & -6 & 1 \\ -5 & 32 & -85 & 120 & -95 & 40 & -7 \\ 17 & -114 & 321 & -484 & 411 & -186 & 35 \\ -15 & 104 & -307 & 496 & -461 & 232 & -49 \\ 137 & -972 & 2970 & -5080 & 5265 & -3132 & 812 \\ -10 & 72 & -225 & 400 & -450 & 360 & -147 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Knuth:

$$Q = \begin{bmatrix} 27 & 9 & 3 & 1 \\ 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ -1 & 1 & -1 & 1 \\ -8 & 4 & -2 & 1 \\ -27 & 9 & -3 & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} 729 & 243 & 81 & 27 & 9 & 3 & 1 \\ 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 64 & -32 & 16 & -8 & 4 & -2 & 1 \\ 729 & -243 & 81 & -27 & 9 & -3 & 1 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 720 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 240 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 144 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 48 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 360 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 60 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R = \begin{bmatrix} 1 & -6 & 15 & -20 & 15 & -6 & 1 \\ 1 & -4 & 5 & 0 & -5 & 4 & -1 \\ -1 & 12 & -39 & 56 & -39 & 12 & -1 \\ -1 & 8 & -13 & 0 & 13 & -8 & 1 \\ 2 & -27 & 270 & -490 & 270 & -27 & 2 \\ 1 & -9 & 45 & 0 & -45 & 9 & -1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

4-way:

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 8 & 4 & 2 & 1 \\ -8 & 4 & -2 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & -2 & 4 & -8 \\ 1 & 2 & 4 & 8 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 64 & -32 & 16 & -8 & 4 & -2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -2 & 4 & -8 & 16 & -32 & 64 \\ 1 & 2 & 4 & 8 & 16 & 32 & 64 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 180 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 120 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 360 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 120 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 180 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -180 & 6 & 2 & -80 & 1 & 3 & -180 \\ -510 & 4 & 4 & 0 & -1 & -1 & 120 \\ 1530 & -27 & -7 & 680 & -7 & -27 & 1530 \\ 120 & -1 & -1 & 0 & 4 & 4 & -510 \\ -180 & 3 & 1 & -80 & 2 & 6 & -180 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

This last solution was implemented as shown in the rather busy Figure 4.

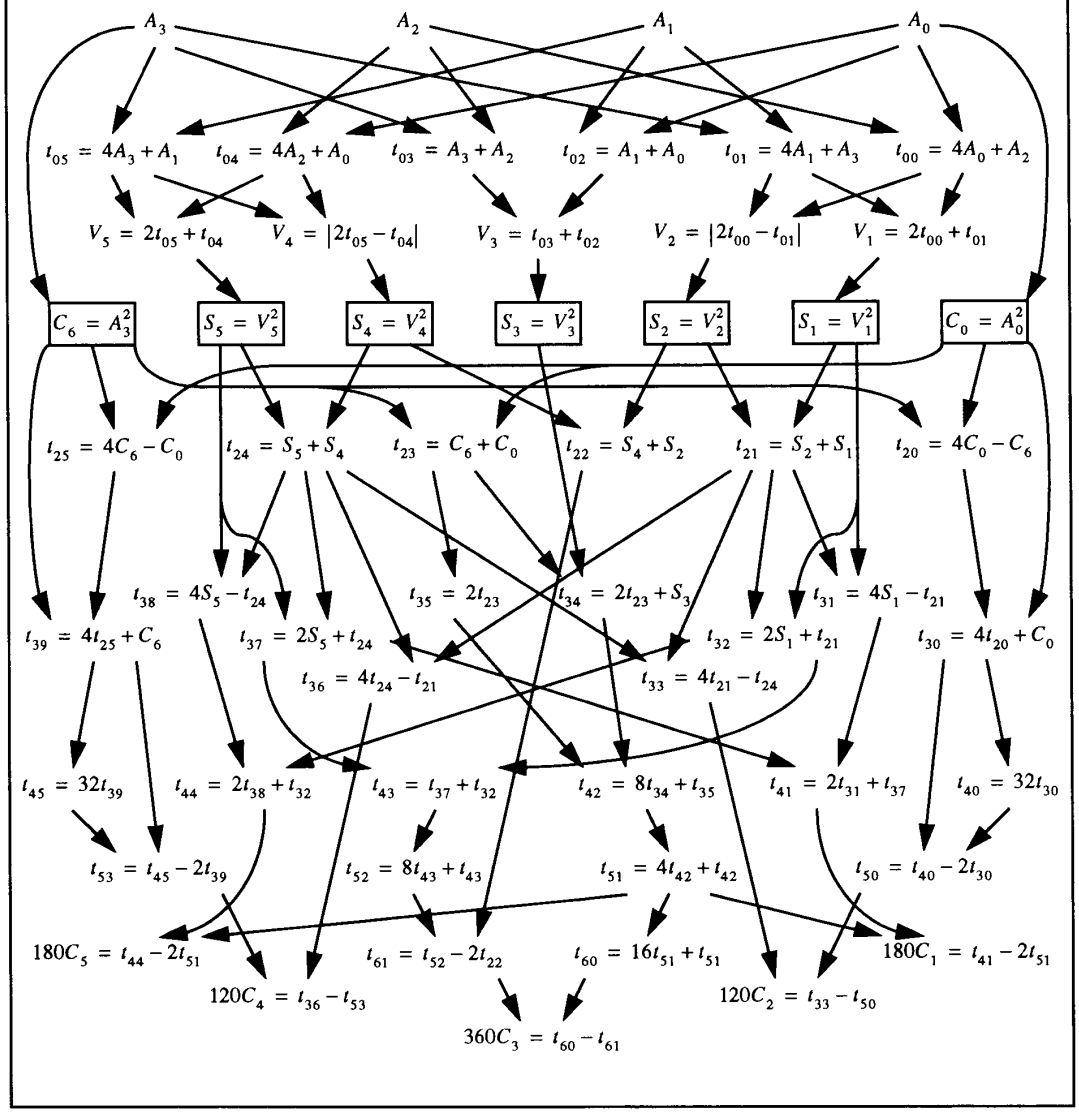
Dividing by -15 (for $120 = 8 \times 15$) can be done by skipping the first step in a divide by -3.

To divide by -45 (for $180 = 4 \times 45$ and $360 = 8 \times 45$), first observe that $4095 = 91 \times 45$ and construct the following chain

$$4\,(45k) + 45k \rightarrow 225k$$

$$8\,(225k) + 225k \rightarrow 2025k$$

Figure 4: The 4-way Method



$2(2025k) + 45k \rightarrow 4095k = (2^{12} - 1)k$

$2^{12}((2^{12} - 1)k) + (2^{12} - 1)k \rightarrow (2^{24} - 1)k$

$2^{24}((2^{24} - 1)k) + (2^{24} - 1)k \rightarrow (2^{48} - 1)k$

$2^{48}((2^{48} - 1)k) + (2^{48} - 1)k \rightarrow (2^{96} - 1)k$

$2^{96}((2^{96} - 1)k) + (2^{96} - 1)k \rightarrow (2^{192} - 1)k$

...

The 4-way method is asymptotically $w^{\log 7/\log 4} \approx w^{1.404}$ and is approximated on the HP-9000/720 by the formula

$$T_4(w) = (1940w^{\log 7/\log 4} - 10709w + 370959)/13$$

10. The FFT Multiply

The best known method of squaring or multiplying integers is the FFT multiply discovered by Schönhage and Strassen[5]. That is, this is the method

with the best known *asymptotic* behavior of $O(w \log w \log\log w)$.

I shall not present this method here in any great detail but I recommend the excellent description in Chapter 7 of Aho, Hopcroft, and Ullman[1].

In brief, the FFT multiply of order k splits a number into 2^{k-1} parts for increasingly large values of k, performs an order k FFT on it, multiplies (or squares) those 2^k elements, and performs an inverse FFT on the result. All arithmetic is performed in a ring with a solution to the equation $\omega^{2^{k-1}} = -1$.

If the ring chosen is the integers mod some base, b, the equation becomes $\omega^{2^{k-1}} = -1 mod(b)$ and b must be larger than the result coefficients.

It is common to choose $b = 2^{m2^{k-1}} + 1$ with $m2^{k-1} > 2w \log w / \log 2$ so that $\omega = 2^m$, although other choices are possible.(This transform is also known as a Number Theoretic Transform or NTT.)

The overhead involved can be large compared to the previous methods but the divides are always by powers of 2.

For example, with $\omega = 2^m$ and $b = 2^{2m} + 1$, we have $\omega^2 = -1 mod(b)$ and an FFT-2 would look like

$$\begin{bmatrix} V_3 \\ V_2 \\ V_1 \\ V_0 \end{bmatrix} = \begin{bmatrix} A_0 + A_1 \\ A_0 - A_1 \\ A_0 + A_1\omega \\ A_0 - A_1\omega \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & \omega \\ 1 & -\omega \end{bmatrix} \begin{bmatrix} A_1 \\ A_0 \end{bmatrix}$$

and

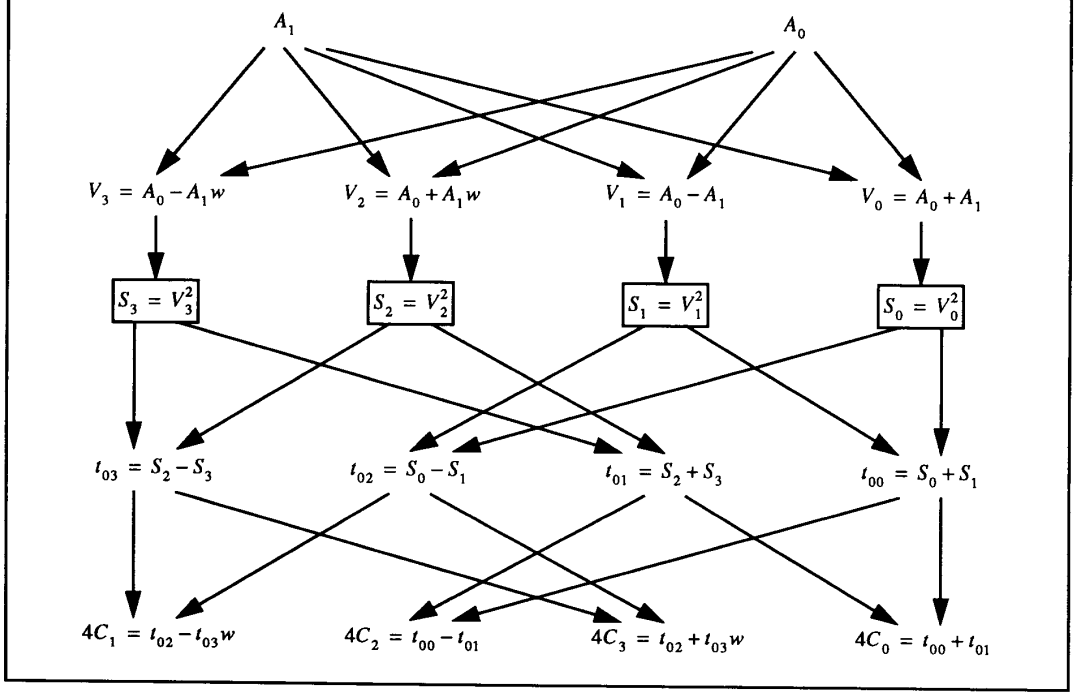
$$\begin{bmatrix} \omega & -1 & -\omega & 1 \\ -\omega & -1 & \omega & 1 \\ -1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix} = \begin{bmatrix} V_3^2 \\ V_2^2 \\ V_1^2 \\ V_0^2 \end{bmatrix}$$

with the solution

$$\begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 \\ 0 & 0 & 4 & 0 \\ 0 & 4 & 0 & 0 \end{bmatrix}^{-1} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & -1+\omega & -\omega \\ 1 & 1 & -1 & -1 \\ 1 & 0 & -1-\omega & \omega \end{bmatrix} \begin{bmatrix} S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix}$$

This method is shown in Figure 5.

Figure 5: The FFT-2 Method



11. Running Time of FFT-*k* Methods

Since b must be large enough to contain the result coefficients, each of the 4 squares in an FFT-2 method must be done with a method which is slightly *larger* than the entire operand! Therefore, this method is not useful for constructing larger squares.

The first method that is useful in that sense is the FFT-3. The operand is divided into 4 parts and 8 squares are performed in a ring that is slightly larger than half the operand. Therefore, this method is asymptotically $O(w^{\log 8/\log 2}) = O(w^3)$!

In general, an FFT-k will have 2^k squares in a ring slightly larger than 2^{k-2} times smaller than the operand. The asymptotic behavior is therefore $O(w^{k/(k-2)}) = O(w^{1+2/(k-2)})$.

A timing formula for an FFT-k may be derived in the same way as the previous methods as follows

$$T_k(w_0) = t_0$$

$$T_k(2^{(r+1)(k-2)}w_0) = 2^k T_k(2^{r(k-2)}w_0) + c_1 k 2^{rk} w_0 + c_0$$

$$T_k(2^{k-2}w_0) = 2^k t_0 + c_1 k w_0 + c_0$$

$$T_k(2^{2(k-2)}w_0) = 2^{2k} t_0 + 2c_1 k w_0 2^k + (2^k + 1)c_0$$

$$T_k(2^{3(k-2)}w_0) = 2^{3k} t_0 + 3c_1 k w_0 2^{2k} + (2^{2k} + 2^k + 1)c_0$$

...

$$T_k(2^{r(k-2)}w_0) = 2^{rk} t_0 + c_1 k w_0 r 2^{rk} + \left(\frac{2^{rk}-1}{2^k-1}\right)c_0$$

Rearranging terms gives

$$T_k(2^{r(k-2)}w_0) = \left(t_0 + 2^{-k} c_1 k w_0 r + \frac{c_0}{2^k-1}\right)2^{rk} - \frac{c_0}{2^k-1}$$

or, with $w = 2^{r(k-2)}w_0$, becomes

$$T_k(w) = \left(t_0 + \frac{c_1 k w_0 \log\frac{w}{w_0}}{2^k(k-1)\log 2} + \frac{c_0}{2^k-1}\right)\left(\frac{w}{w_0}\right)^{\frac{k}{(k-2)}} - \frac{c_0}{2^k-1}$$

12. Some Actual Results

Figure 6 compares the methods discussed here.

As a log-log plot, Figure 6 diminishes the differences between the various methods. We can illustrate those differences better by rescaling the data by $T_2(w)$.

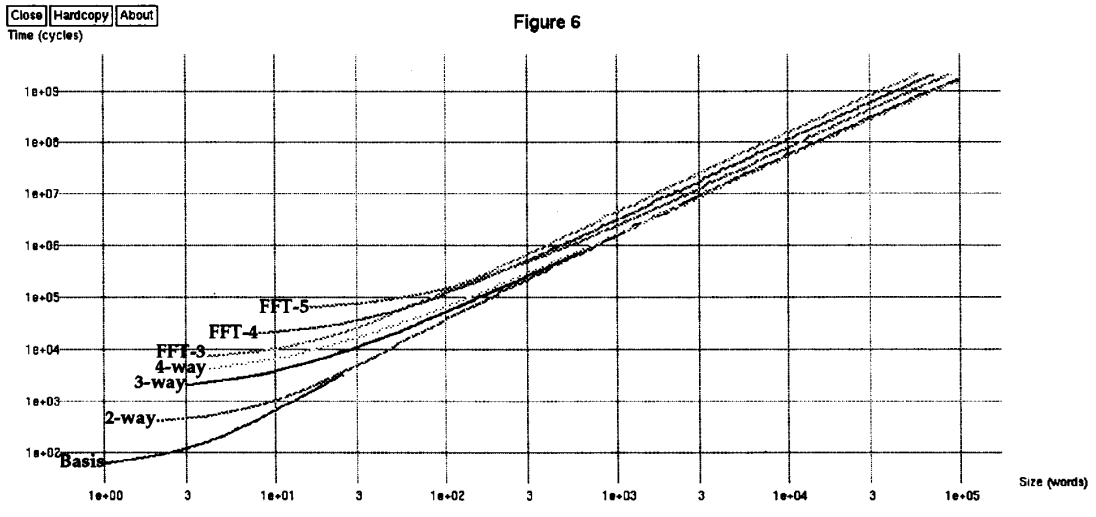
Figure 7 shows a log-linear plot of $T_i(w)/T_2(w)$. We can now see that the k-way methods are faster than the FFT-k methods for $w < 10^5$ words (3×10^6 bits) but we still don't have a clear idea of the asymptotic behavior.

For that, the ratios

$$\frac{\log\frac{T_k(w)}{T_k(w/k)}}{\log k}$$

for the k-way methods and

Figure 6



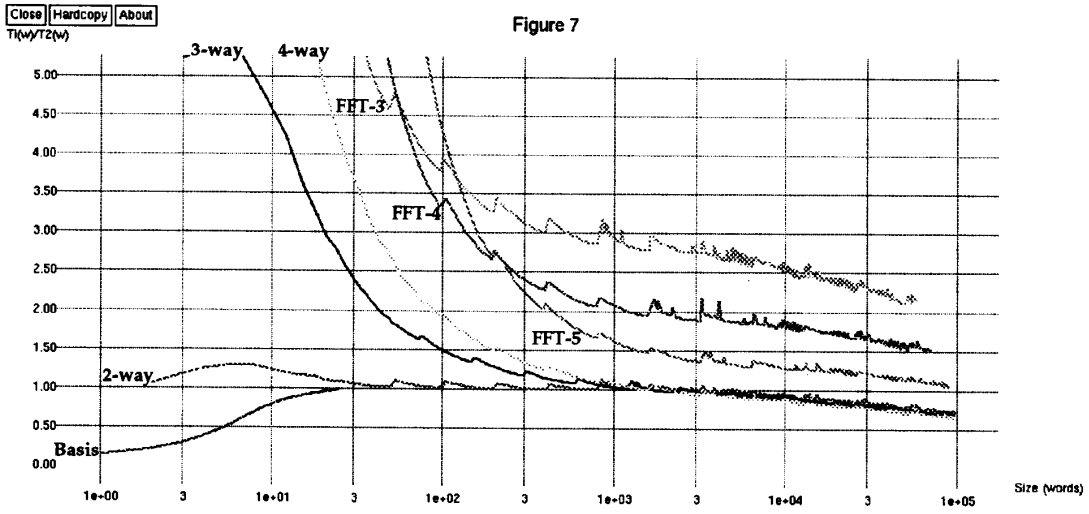


Figure 7

$$\frac{\log \dfrac{T_k(w)}{T_k(w/2^{k-2})}}{(k-2)\log 2}$$

for the FFT-k methods will illustrate how quickly each approaches its asymptote. Figure 8 shows this.

None of the FFT-k methods tried have any chance of being best. An FFT-6, asymptotically $O(w^{\log 64/\log 16}) = O(w^{1.5})$, will also be too slow. An FFT-7, $O(w^{\log 128/\log 32}) = O(w^{1.4})$, is asymptotically faster than the 4-way method but the 5-way method, $O(w^{\log 9/\log 5}) = O(w^{1.365})$, very likely has much less overhead and will win first.

In spite of this behavior for small orders, it is possible that the FFT-k methods will eventually win for some $w \gg 100,000$ words.

The overhead in an FFT-k is known to grow as something like $O(k2^k)$. A k_{TC}-way method is asymptotically comparable to an FFT-k_{SS} method when $k_{TC} \sim 2^{\frac{k_{SS}-2}{2}}$. If the overhead of the k-way methods, dominated by a matrix inversion, grows as fast as $O(k_{TC}^3) \sim O(2^{\frac{3}{2}(k_{SS}-2)})$, the FFT-$k$'s may eventually win.

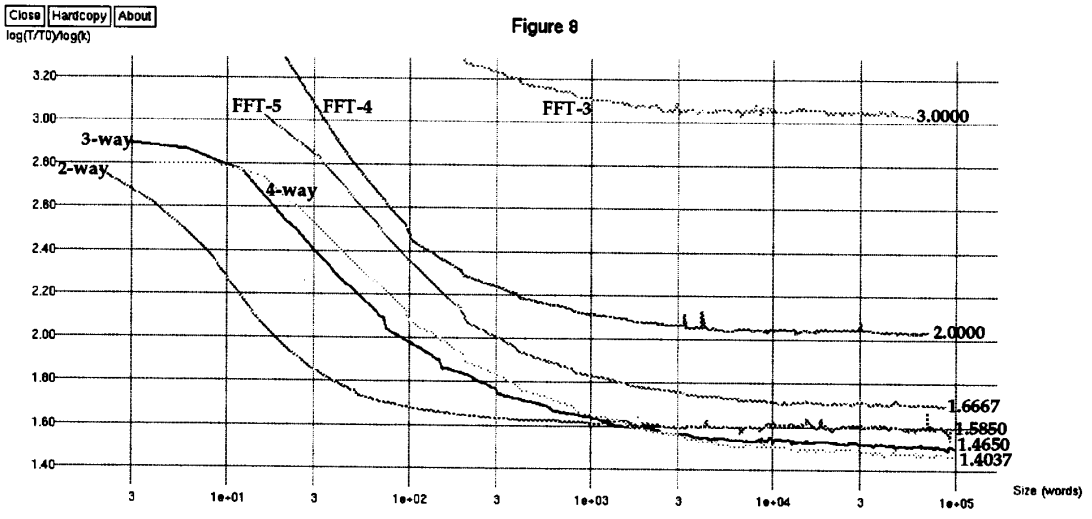


Figure 8

13. Is Squaring Faster than Multiplying?

Squaring is a special case of multiplying. Therefore, we have trivially that

$$T_{square} \leq T_{multiply}$$

But, in all the methods presented here, squaring involves *strictly* less work than multiply. Further, most of this savings is in the overhead and, in the limit of large numbers, virtually all of the work in a multiply is in the overhead.

Therefore, we are led to ask the question: Is it possible that there are squaring methods that are *of an order* faster than any multiply methods?

The answer is, unfortunately: no.

While it is possible that we may discover some method of squaring that is strictly faster than any *existing* multiply method, any squaring method can be used to construct a multiply method that is no more than a constant slower.

A simple proof is

$$XY = ((X+Y)^2 - (X^2+Y^2))/2$$

which, assuming that add and shift are no worse than $O(n)$, shows that

$$T_{multiply} \leq 3T_{square} + O(n)$$

Karatsuba presented a better method

$$XY = ((X+Y)^2 - (X-Y)^2)/4$$

which gives a multiply in 2 squares and $O(n)$. Therefore,

$$T_{multiply} \leq 2T_{square} + O(n)$$

14. Conclusions, Speculations, Etc.

It would appear that many of the simpler methods of multiplying are best all the way out to quite large numbers. Certainly, into the millions of bits. Possibly, much farther.

In spite of the fact that squaring is fundamentally faster than multiply, it can be no better than a constant faster in the limit of large numbers.

It is still possible that the Schönhage and Strassen method will win in the end in spite of a slight asymptotic disadvantage. This would be a useful area for further work.

A closely related area is that of what is the minimum amount of overhead possible in Toom-Cook style methods. Different assumptions about the cost of overhead might lead to different trade-offs.

For example, in the approach taken in this paper, what are the best choices of $\pm\frac{p}{q}$ so as to minimize add, shift, and divide overhead?

15. Acknowledgments

I would like to thank the reviewers for their comments. I believe the resulting changes improved the paper considerably.

I would also like to thank Willy McAllister for his support and Cathrin Callas for her help in the preparation of this paper.

Many of the figures in this paper were made with *xgraph*, a program written by David Harrison of the University of California.

16. References

1. Aho, A.V., Hopcroft J.E., Ullman J.D., *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974, Chapter 7.

2. Cook, S.A. *On the minimum computation time of functions*, Thesis, Harvard University, May 1966, pages 51-77.

3. Karatsuba, A. and Ofman, Yu. *Multiplication of Multidigit Numbers on Automata*, Soviet Physics - Doklady, Vol. 7, #7, January 1963, pages 595-596.

4. Knuth, D.E. *The Art of Computer Programming, Vol. 2.*, Second Edition, Addison-Wesley, Reading, Mass., 1981, Chapter 4, Section 3.3, pages 278-301.

5. Schönhage, A. and Strassen, V. *Computing* 7 (1971), 281-292 (in German).

6. Toom, A.L. *The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers*, Soviet Mathematics, Vol. 3, 1963, pages 714-716.